SLAM – part 2
Empirical Analysis

EDAA – G06

João Costa – João Martins
Henrique Ribeiro
Tiago Duarte

Why sets

- Brief C++ history
- Rationale
- Set concepts

Rationale

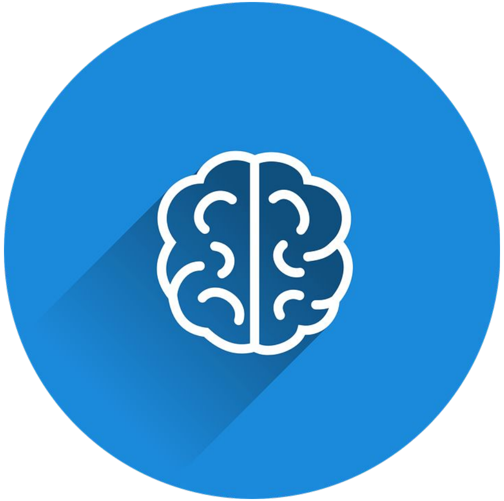
→ Related to C++ inception

Rationale – Brief C++ history

- Back when C++ was being standardized:
  - **Open hashing** was still a new topic (not mature enough);
  - C++ sets and maps were based on a paper from 2003 by Matt Austern, which leveraged **closed hashing**;
  - This was the safe approach at the time.
- At the beginning, the API didn't require the implementation to use **closed hashing**:
  - From C++17 onwards, with the introduction of "*extract*" capabilities, it became required.
- The API requires the following:
  - Iterator increment must be (amortized) constant time;
  - The erase method must be constant time on average.

Rationale – Opportunities for improvements

- Our use case needs unordered sets as containers for very small objects:
  - Objects are 48-bit < 64-bit pointers;
  - C++ sets are for general usage, so they are optimized for big objects (> 64-bit);
  - We can leverage the better cache locality of **open hashing**.
- C++ sets use buckets:
  - Increases memory usage;
  - Nodes needs an extra pointer to the next object (using more memory).
- *N* is the number of elements. *B* is the number of buckets:
  - 16 *N* + 8 *B* (hash caching);
  - 8 *N* + 8 *B* (no hash caching).

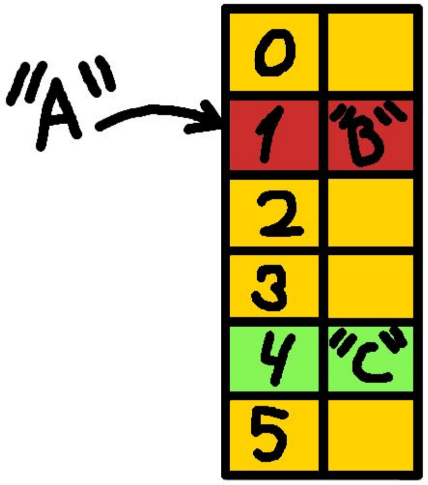


Collision resolution strategies

→ A recap

What happens during an insert

- **Set** data structures are supported by **hash tables**;
- During the insertion operations:
  - We calculate the **hash** of the object;
  - From the **hash** we obtain an **index**;
  - The element is stored at the **index**.
- It is possible to obtain the same **hash/index** from an object:
  - When this happens, we have a **collision**.

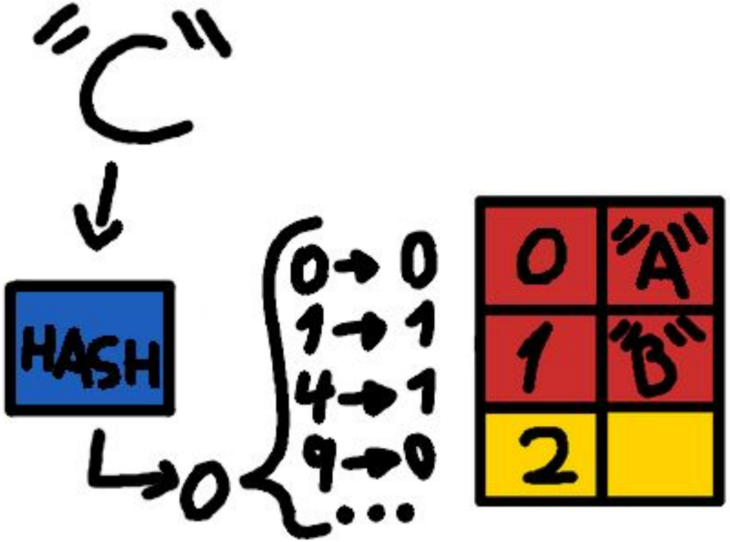


Collision resolution strategies

- One of the simplest ways to deal with collisions is **closed hashing**:
  - Each index is a **bucket**;
  - A bucket is a linked list of elements with the same hash;
  - This has performance problems.
- The group implemented 3 **open hashing** techniques:
  - **Linear probing** – on collision, we check the next buckets (one-by-one) until we find a free one;
  - **Quadratic probing** – similar to *linear probing*. We check buckets in jumps of *power-of-two* size, e.g.,: 1 2 4 8...
  - **Double hashing** – we have 2 hash functions: 1 for the object hash and another for the offset.
- **Future work:**
  - Try **Cuckoo hashing** and **Robin Hood hashing**;
  - Apply fast modulo: https://arxiv.org/abs/1902.01961

Implementation details – Quadratic probing

- **Quadratic probing** as described can lead to cycles;
- The basis of the solution to this problem is: keep the **hash table size** a **power of 2**;
- The **quadratic probing** offset becomes: $\mathbf{i^2 \Rightarrow (i^2 + i)/2}$;
- These guarantee that a free slot can always be found.



Implementation details - Double Hashing

- **Double hashing** can lead to cycles;
- We can’t use the same hash function to hash the element and to calculate the offset;
- The cycle size is the **LCM** of the result of the **offset function** and the **hash table size**;
- If these 2 numbers aren’t relatively prime, this number can be significantly low:
  - This is achievable by making one of the numbers prime: i.e., the table size;
  - This has a significant **overhead** for **set resizing**, and the **modulo operation**;
  - The other approach is keeping the **hash table size** a **power of 2** and guarantee that the offset always returns an **odd number**.

Implementation details – Set **dynamic resizing**

- When resizing the hash table:
  - We create a **new one** with double the size;
  - **Re-insert all elements** into this new table.
- This process uses an alternative insert function (move function) optimized for this operation:
  - Reduces the overhead of memory management;
  - Performs fewer checks: we know there aren't tombstones;
  - All hash values are cached (no need to re-calculate them).
- Resizing is triggered when an insert causes the **load factor** to go over **75%**: i.e., 75% of the hash table contains an element (not a tombstone);
- **Future work:**
  - Explore other dynamic resizing methods that explore possibilities that aren't **all-at-once**.

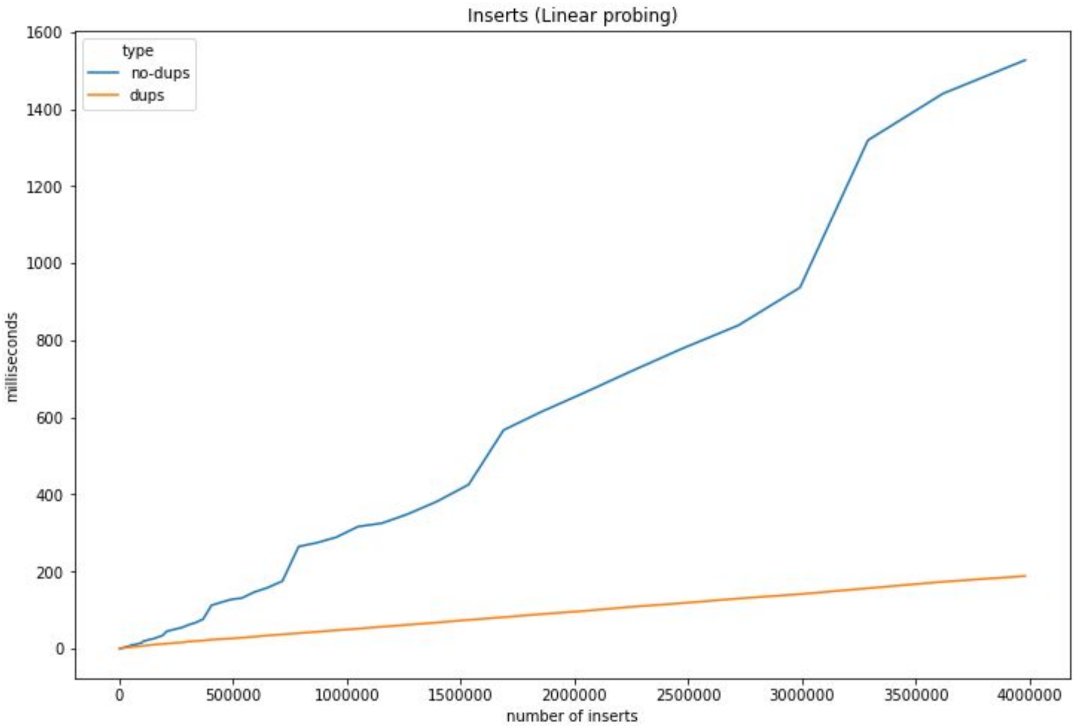
Set performance analysis

- Set operation’s performance analysis
- Collision resolution strategy comparison

Insert operation

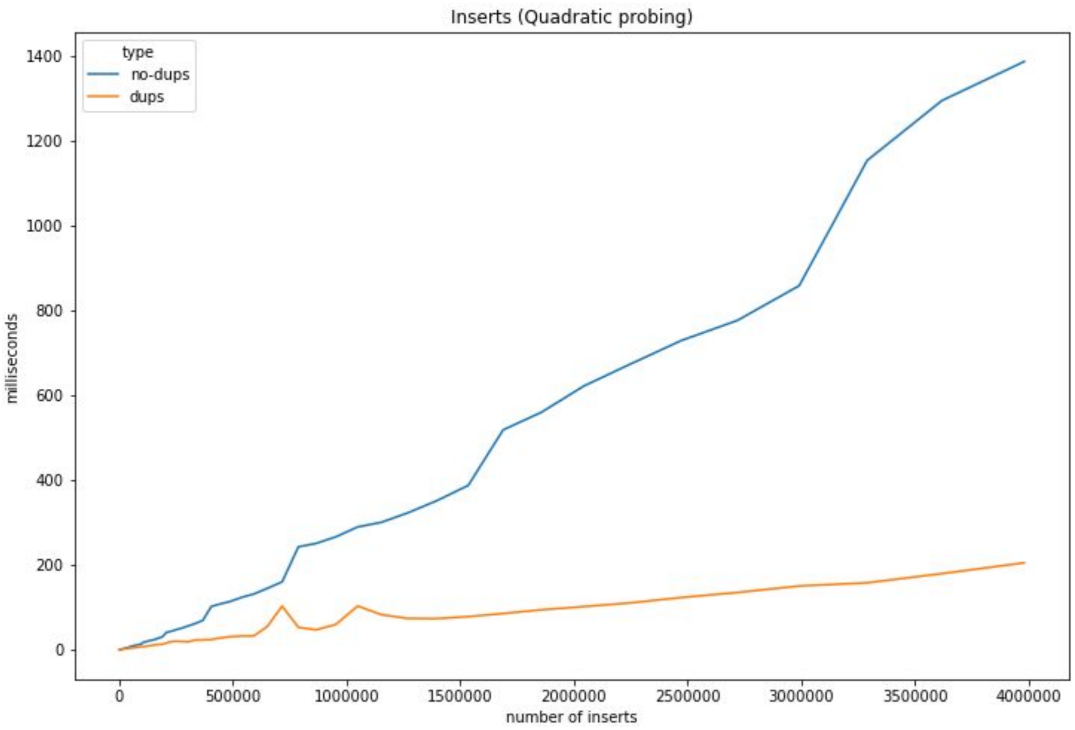
→ Performance analysis

Insert – Linear probing



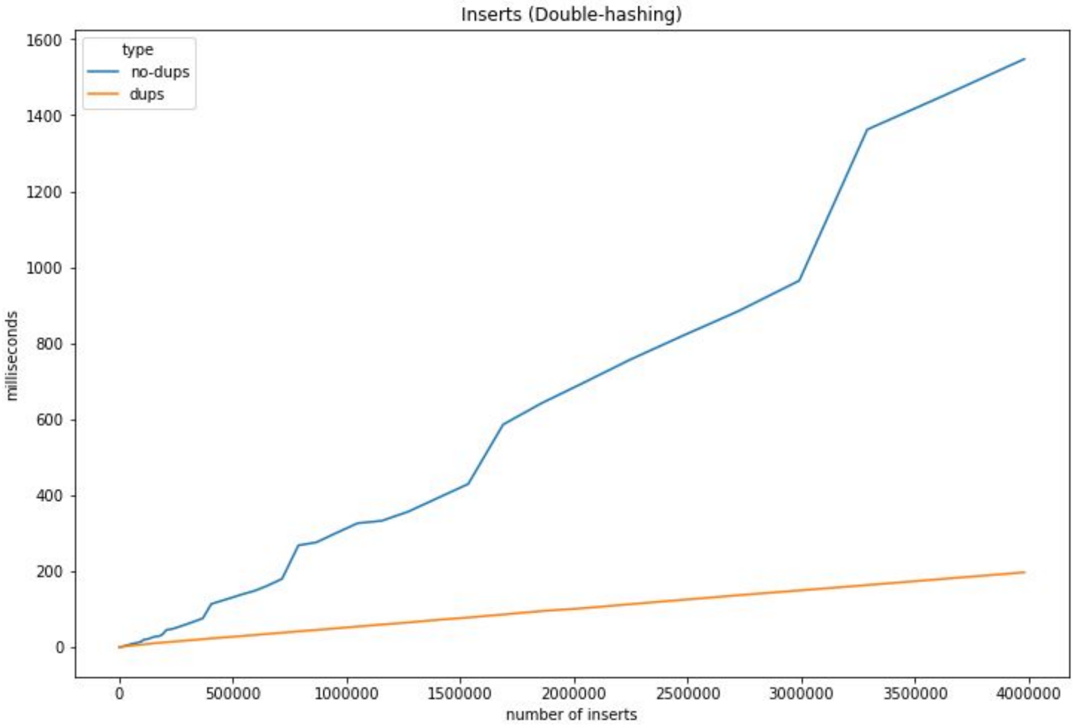
- Insertion of randomly generated objects;
- Collision resolution using **Linear probing**;
- **Blue line** – ≃ **0** duplicated objects;
- **Orange line** – ≃ **58%** duplicated insertions;

Insert – Quadratic probing



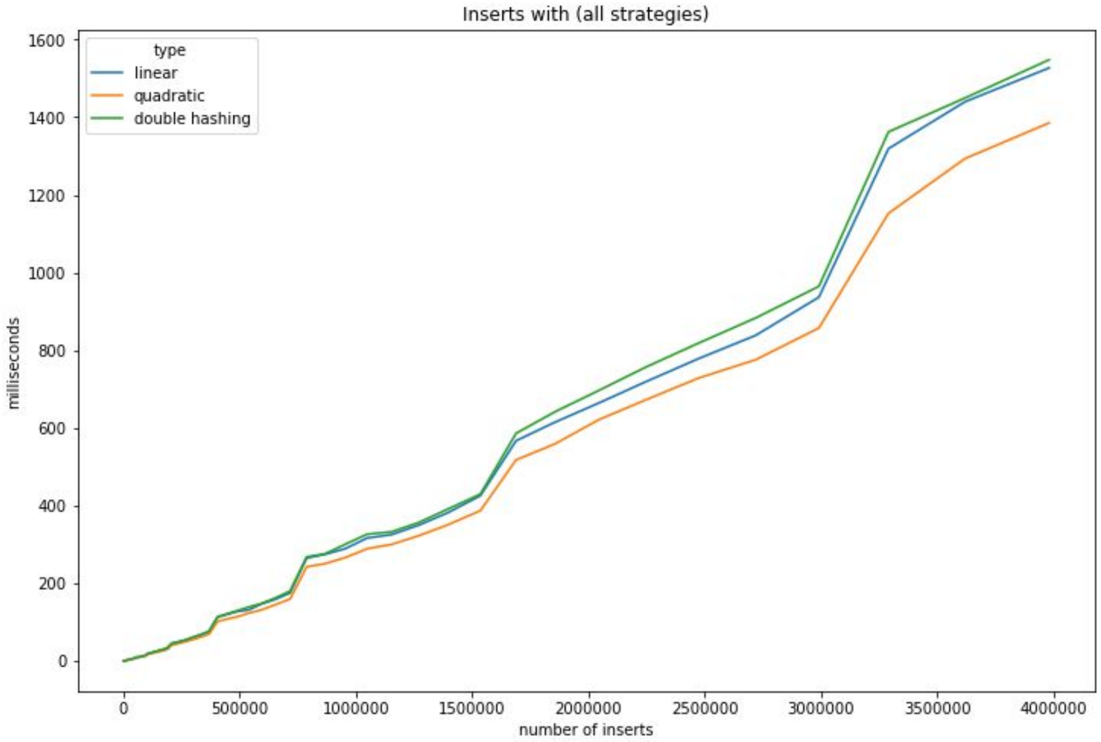
- Insertion of randomly generated objects;
- Collision resolution using **Quadratic probing**;
- **Blue line** – ≃ **0** duplicated objects;
- **Orange line** – ≃ **58%** duplicated insertions;

Insert – Double hashing



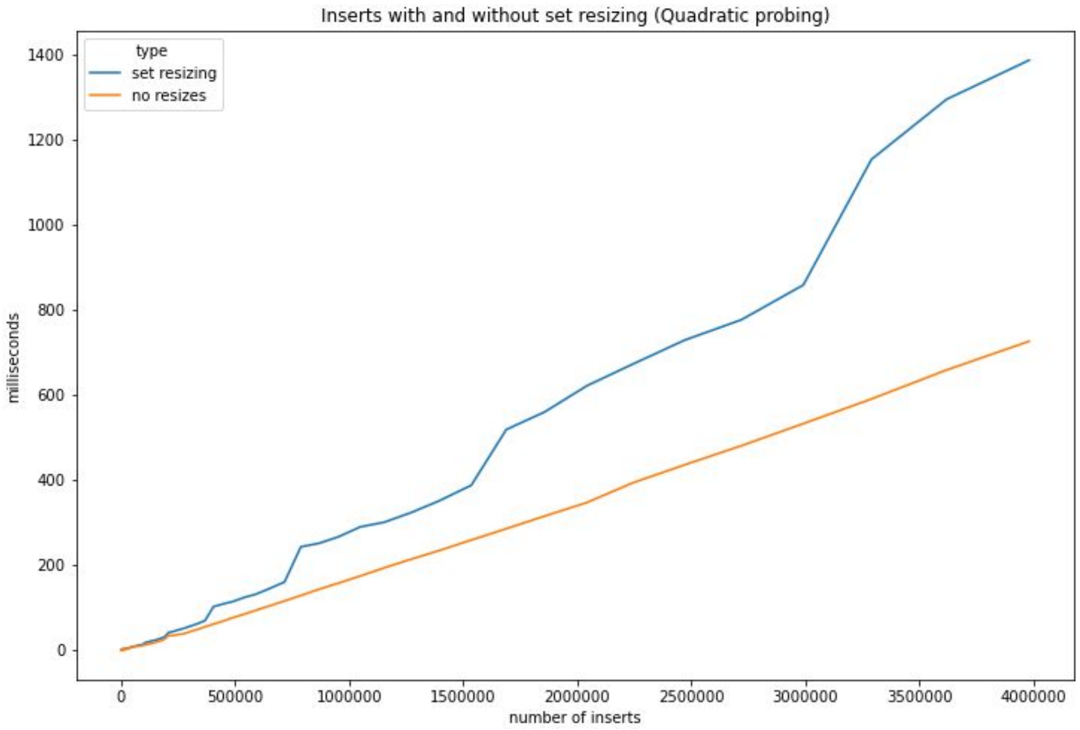
- Insertion of randomly generated objects;
- Collision resolution using **Double hashing**;
- **Blue line** – ≃ **0** duplicated objects;
- **Orange line** – ≃ **58%** duplicated insertions;

Insert – Comparison of all strategies



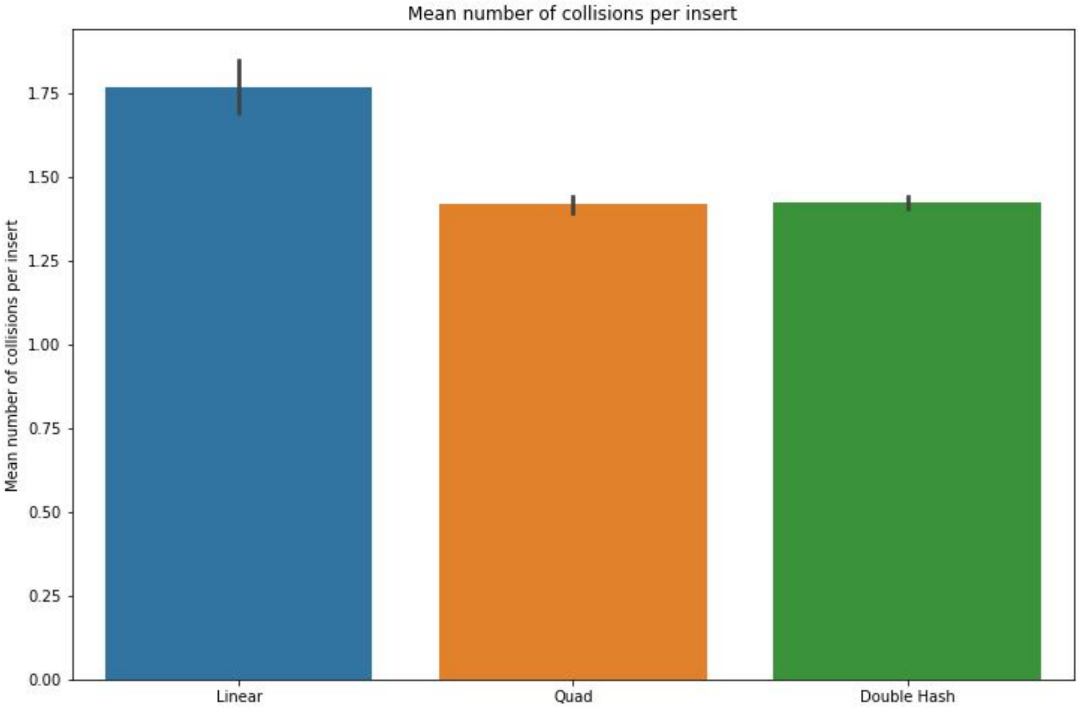
- Insertion of randomly generated objects;
- Performance is similar between all methods;
- **Quadratic probing** is consistently the best.

Insert – Impact of set resizing



- In the previous examples, the sets were resized multiple times;
- This also affects the comparisons:
  - Smaller sets suffer fewer resizes.
- Resize is triggered by a threshold of how full the set is (**75%**);

Insert – Number of collisions



- **Quadratic probing** has the best performance:
  - In par with the fact that it suffers a low amount of collisions.
- **Double hashing** delivers on the promised reduced number of collisions, but performance is hurt:
  - It as the worst performance.

Lookup/Delete operation
→ Performance analysis

Lookup/Delete – Performance

- Lookups and deletes are very similar operations in our set implementation:
    - A *delete operation* is a lookup that sets the entry as dead instead of returning it.
- Lookups for **non-existent** items on big sets are fast:
    - Because of the load factor control, there are numerous empty buckets ⇒ the search is quick to end.

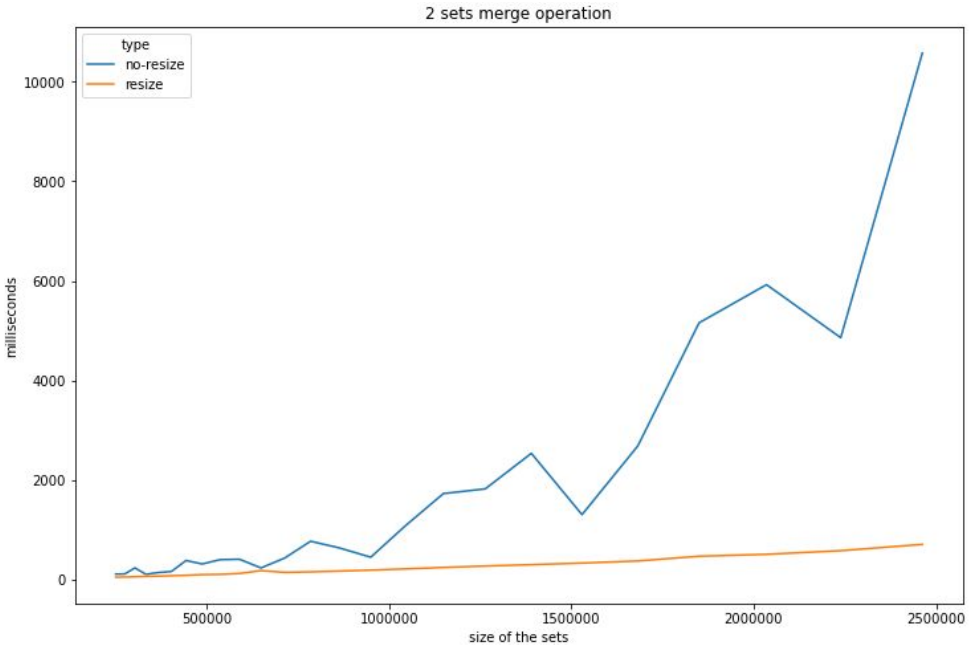
	Existing	**Non-existent**
100000	≃ 6ms	≃ 3ms
200000	≃ 15ms	≃ 5ms
300000	≃ 23ms	≃ 5ms
400000	≃ 32ms	≃ 6ms
500000	≃ 34ms	≃ 8ms
600000	≃ 42ms	≃ 9ms

Merge operation

→ Performance analysis

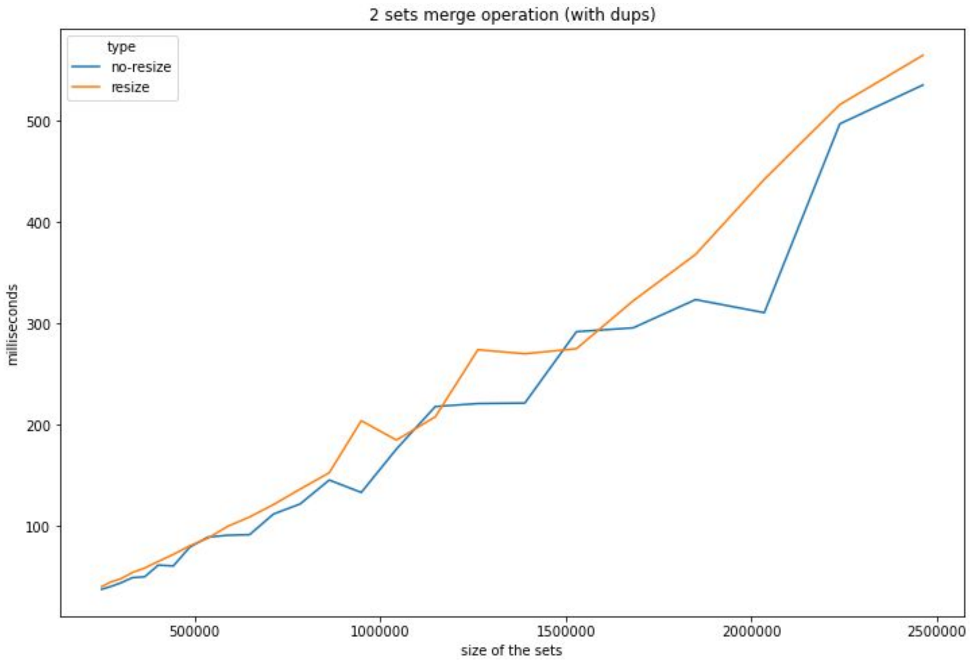
Merge – Performance and impact of resize

- In our project, we only need **in-place merges**;
- The **merge** operation consists of iterating over a set and inserting each object into the second one;
- By doing a **pessimistic assumption**, we can greatly increase the performance of the operation:
  - Assume that each element in the second set is new (not already part of the set);
  - Check that the **load factor is still ok** in that case (no resize needed).

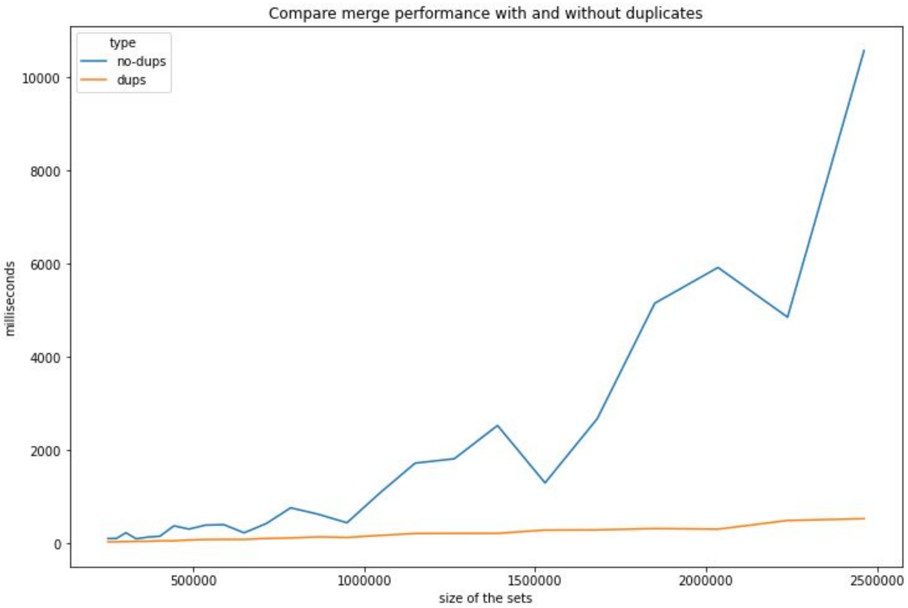


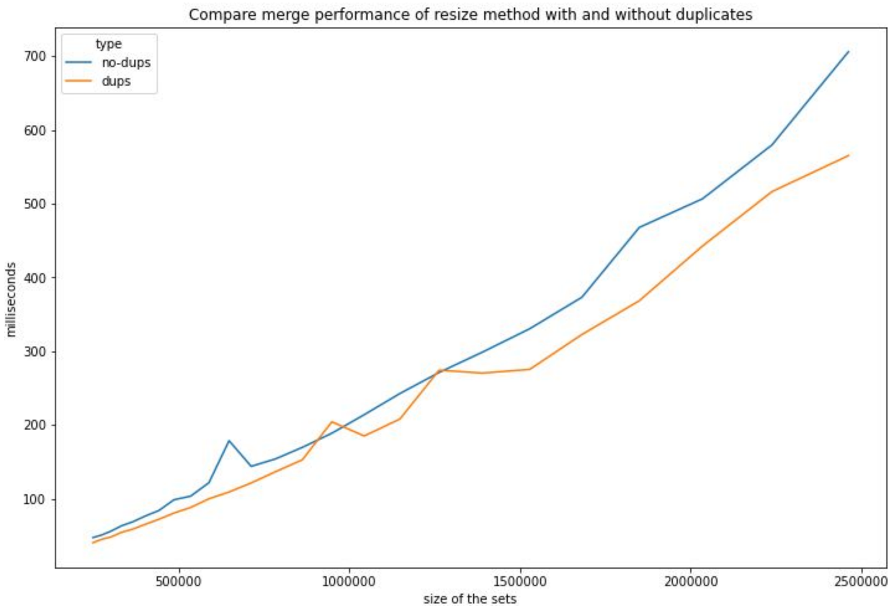
Merge – Optimization with numerous duplicates

- Our use case leads to merges with numerous duplicate entries;
- In this test, the second set has **≃50% of its elements in common** with the first set;
- The optimization doesn't bring anything to the table performance-wise;
  - But can cause an **increase in memory consumption**.



Merge – The impact of duplicates

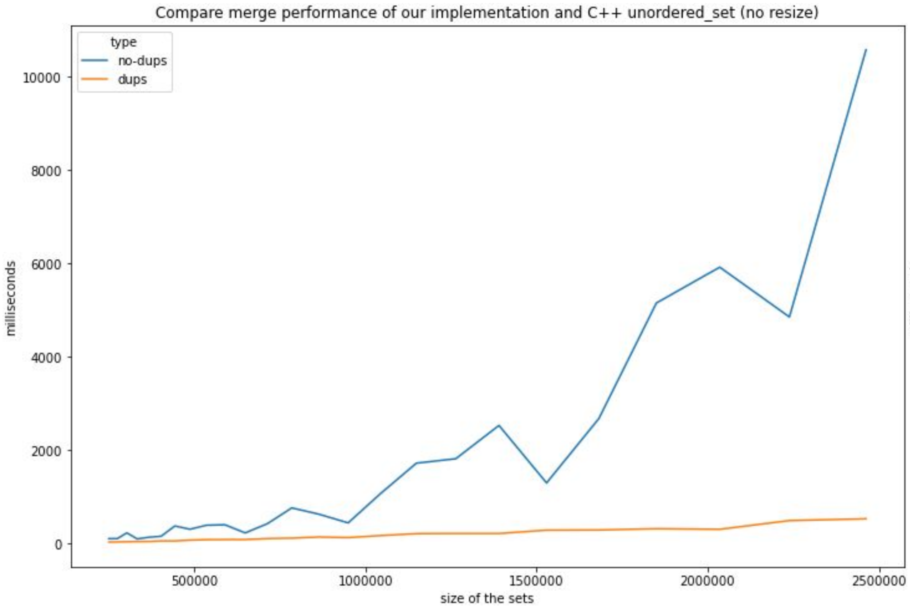


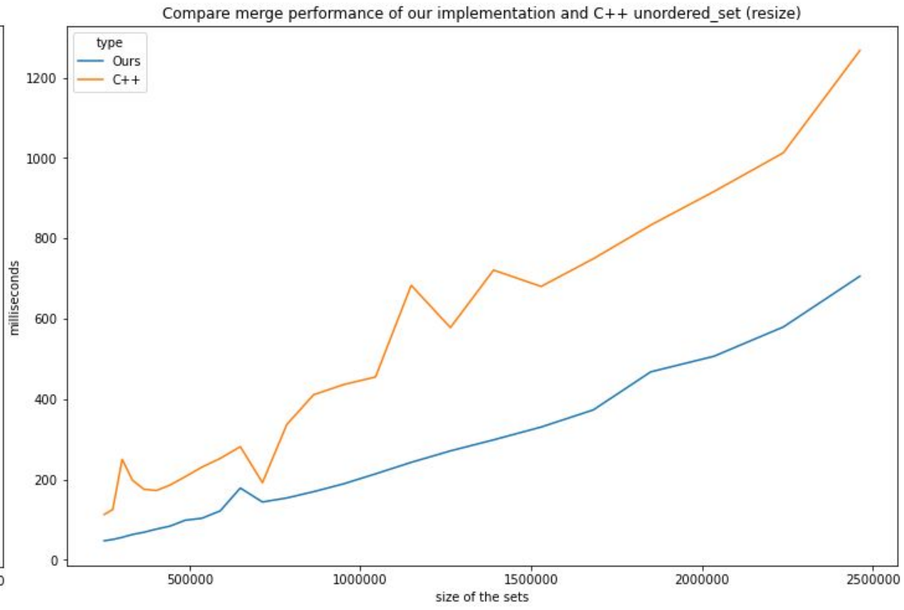


Merge operation

→ Comparison with C++

Merge – Comparison with C++ unordered_set



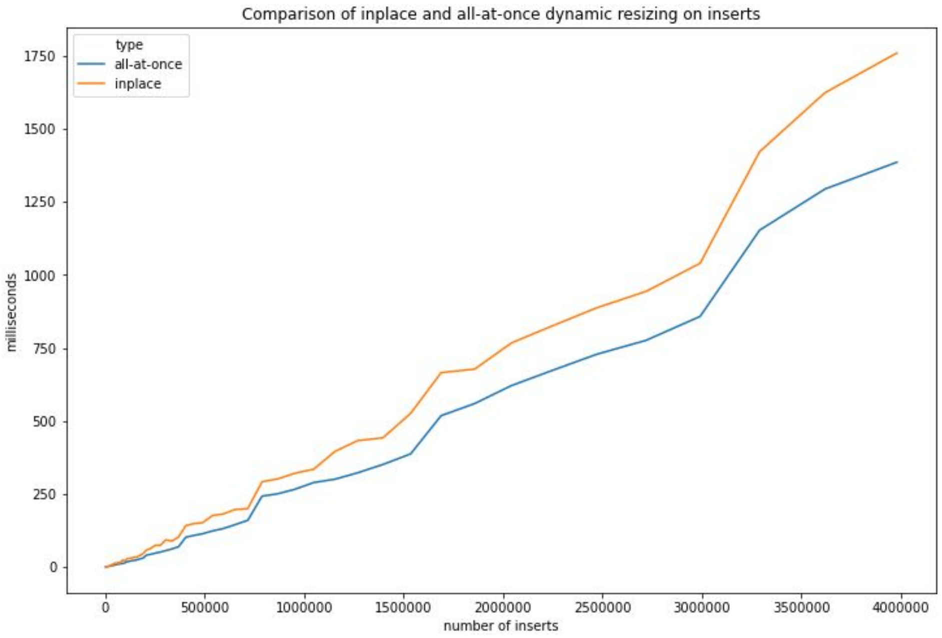


In-place dynamic resizing

→ Comparison with all-at-once

In-place dynamic resizing – comparison

- We implemented a way for sets to be dynamically resized in-place:
  - Less memory overhead;
  - Less time performance.
- Algorithm:
  - Grow the container (double size);
  - Re-evaluate all entries in first half;
  - If they fall on second half, place them there;
  - Otherwise, save them in a buffer for later;
  - Clear first half and insert the buffer elements



Real data test

→ Performance on the dataset

Real data test

- Tests on real data:
  - Ray-casts on a plane point cloud dataset;
  - Only considering the turbines;
  - About 1000 ray-casts covering 18000 cells each.
- Ray-casts are calculated 4 at a time in parallel:
  - On a computer with 4 CPU cores.
- Our set implementation **improved performance almost 2-fold**.

	C++ sets	**Our sets**	**In-place resize**
Time taken	≃ 23 s	≃ 13 s	**≃ 10 s**

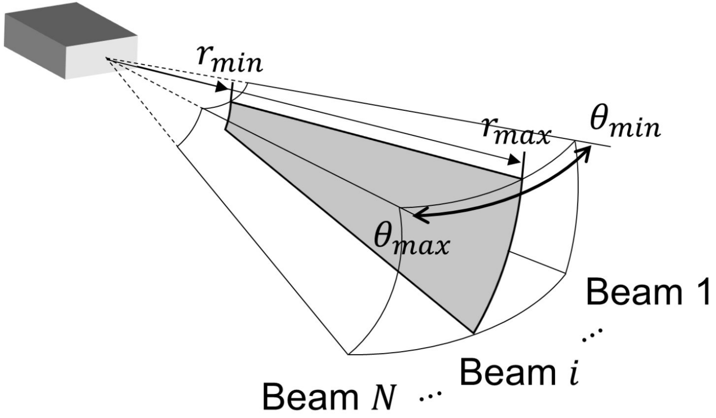
3D Scan Analysis

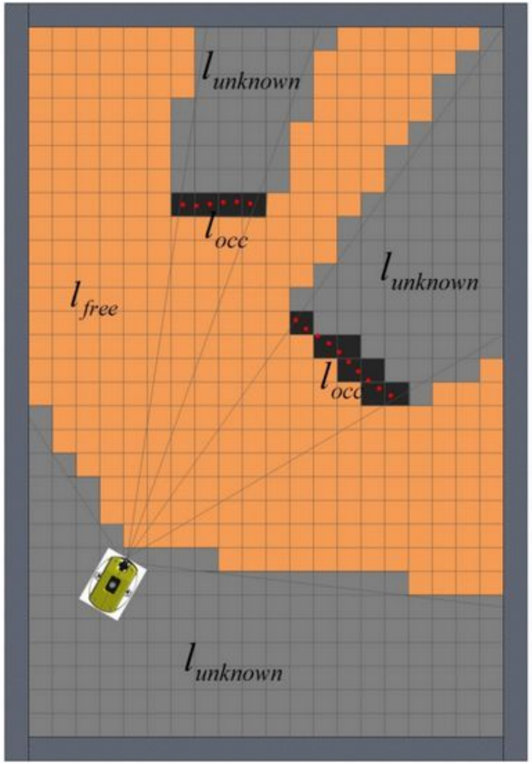
- Cell coverage analysis
- Time analysis

3D Scan Analysis
→ Quick Recap

Sonar Data & 2D Mapping

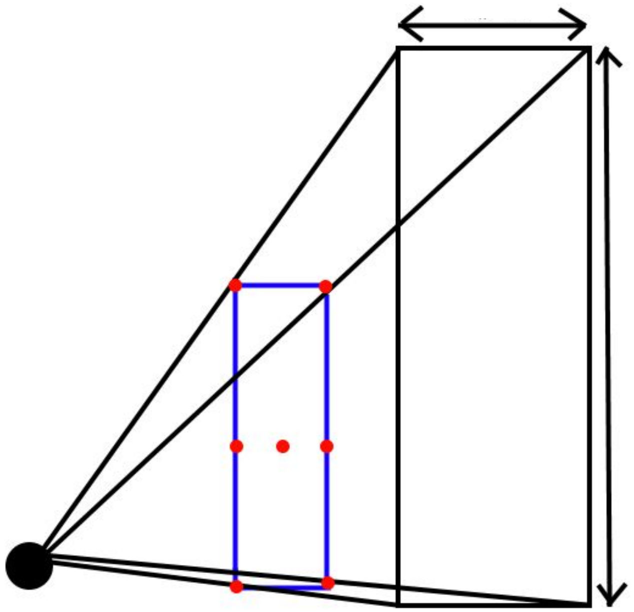
- Sonar rotates around itself;
- Sends/measures waves in a cone;
- Each beam has **multiple intensities across several intervals**.





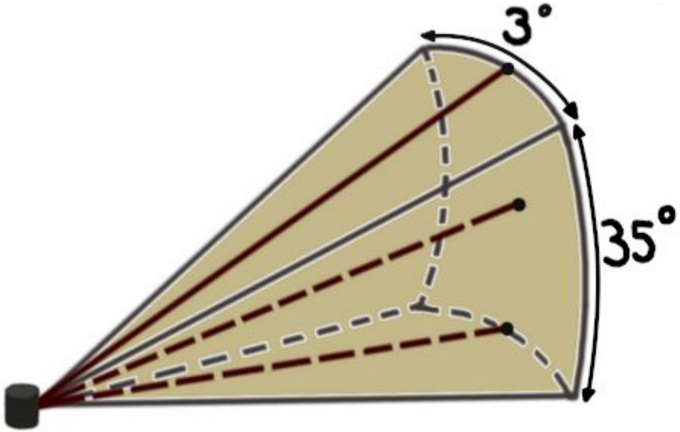
3D - How to Cover a Volume?

- Estimate covered cells through 3D ray-casts
- Choose destination points to achieve maximum coverage
- Divide both horizontal and vertical plane
  - For the same AUV, Fula[1] uses 7 rays
  - But what is the optimal number of divisions to maximize coverage?
  - And what is their impact on performance?



Sonar Beam Spread

- Each sonar beam covers an area that spans over **3º horizontally** and **35º vertically**;
- As such, the vertical plane needs to be further divided to capture the same level of detail.

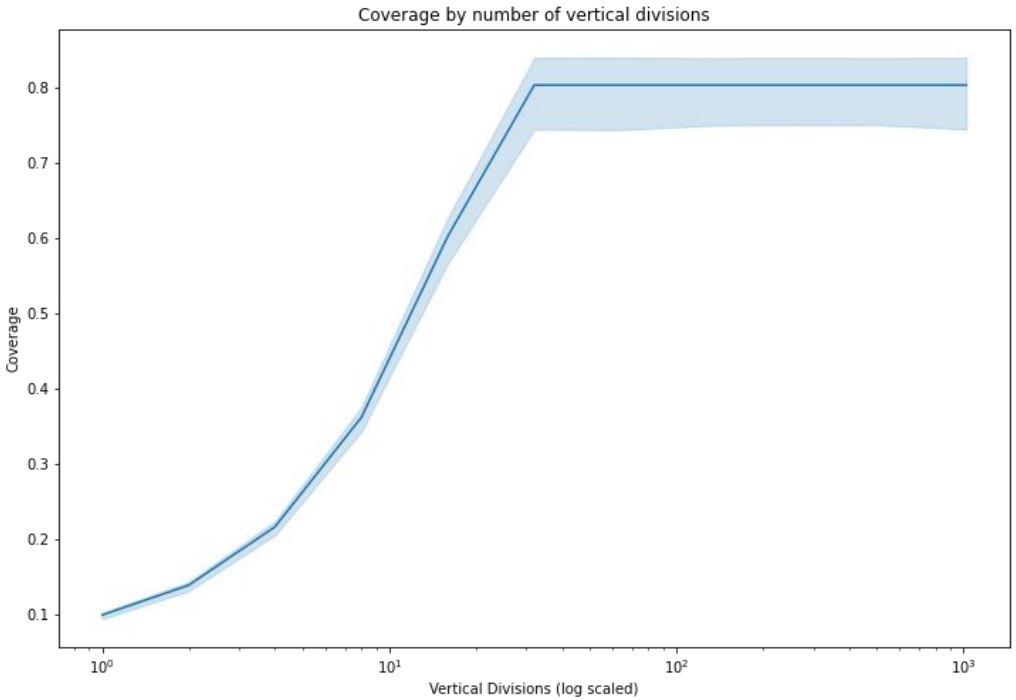


3D Scan Analysis

→ Division Analysis

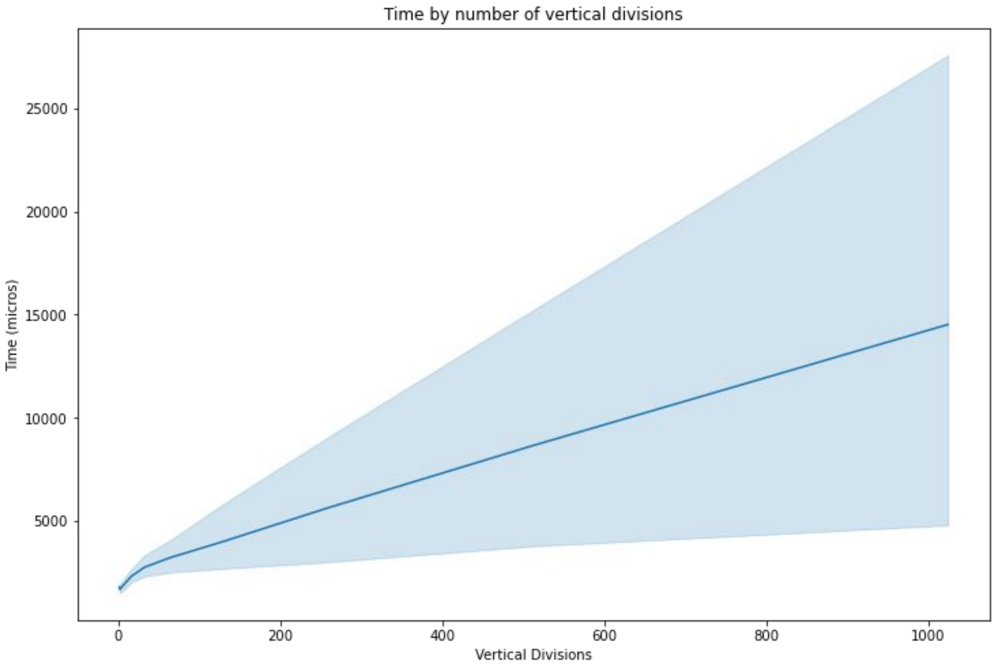
Vertical Divisions – Coverage

- Results were as expected;
- Coverage increases **linearly** until 64 divisions;
- Subdividing the plane further doesn't lead to more coverage.



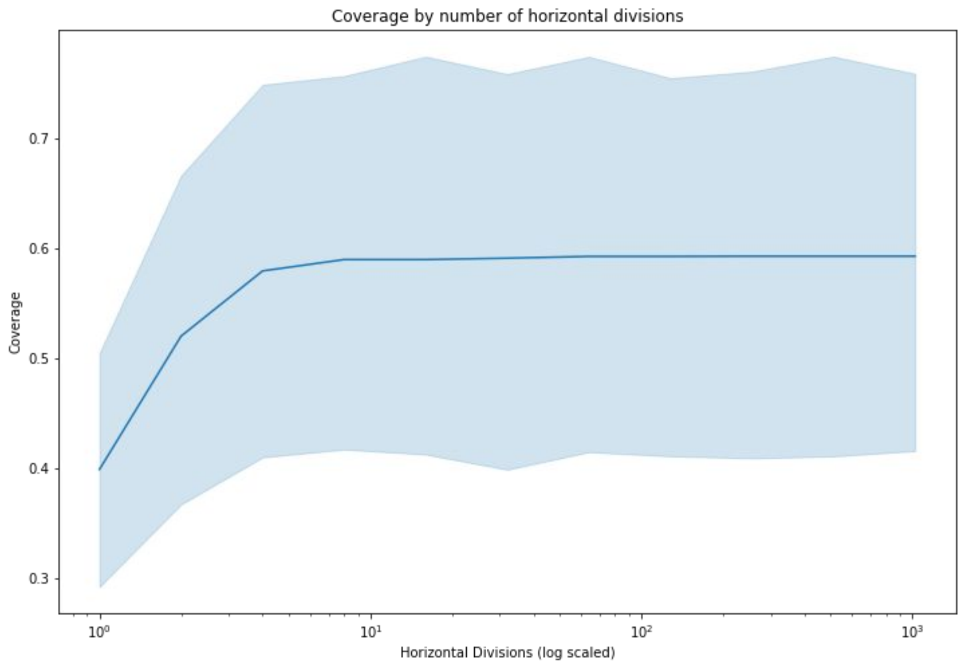
Vertical Divisions – Time

- Time increases linearly;
- Even though our point-cloud implementation is $O(N^2)$;
- We believe that this is due to most ray casts having a lot of points in common:
  - This leads to fast merges;
  - This is why the merge operation is extremely important.



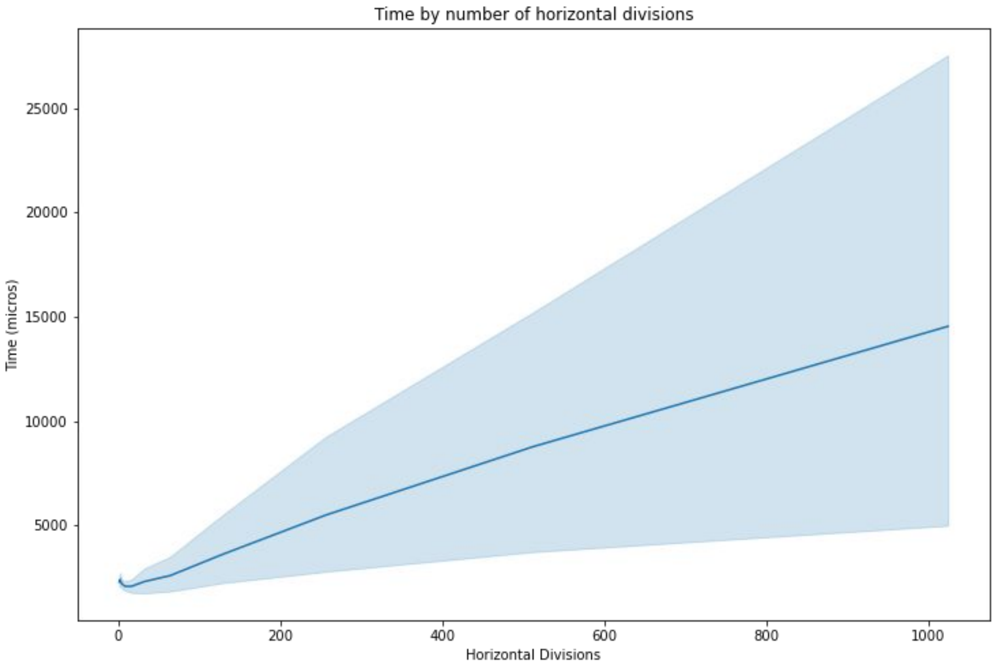
Horizontal Divisions – Coverage

- Results were also as expected;
- Maximum coverage is reached sooner;
- Subdividing the plane further from 32 divisions doesn't lead to more coverage.



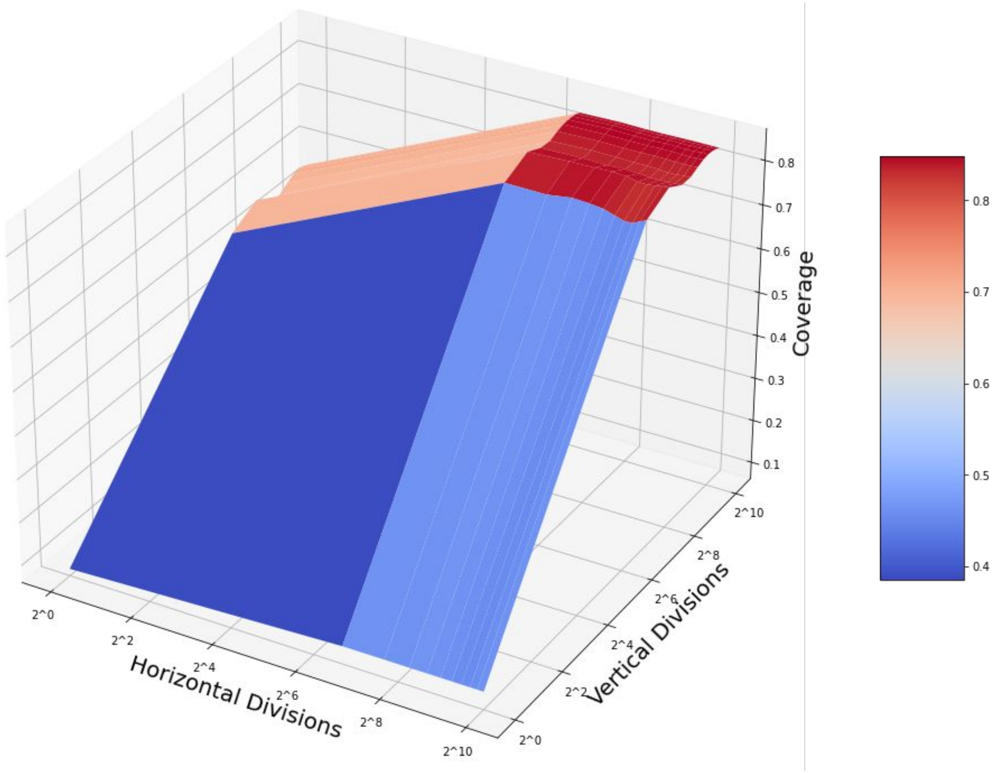
Horizontal Divisions - Time

- Time also increases linearly;
- Same conclusion as vertical performance.

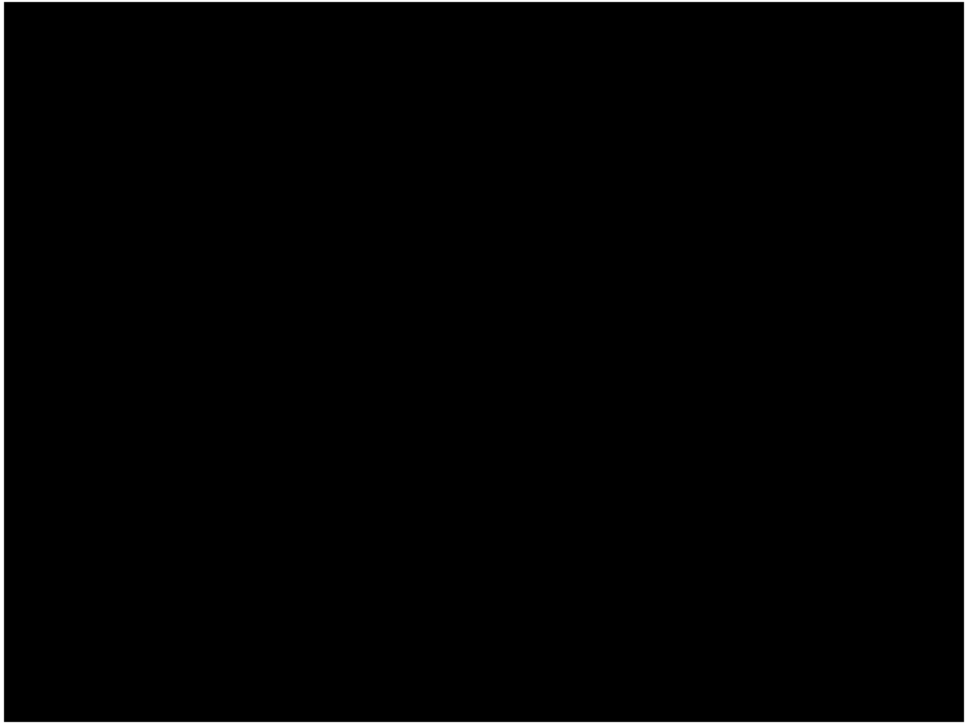


Multivariable Analysis – Coverage

- High coverage is only possible with high horizontal and vertical Divisions;
- These parameters don't seem to affect performance by much:
    - At least in our dataset;
    - So this won't be much of a factor in our decision.

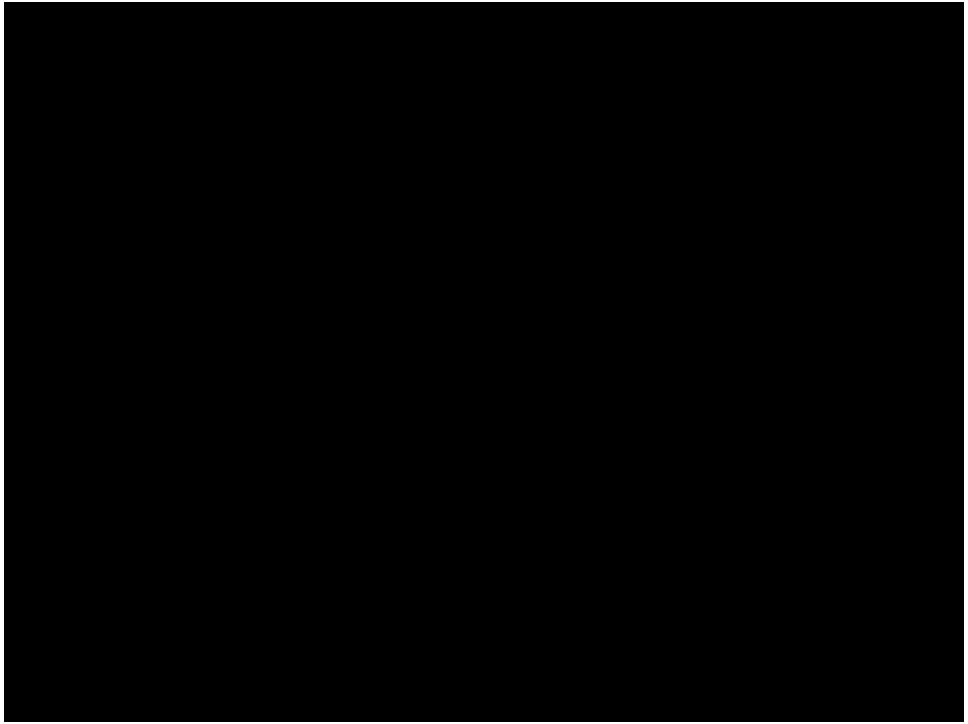


Number of Divisions – Results



1 horizontal and 2 vertical Division

Number of Divisions – Results



64 horizontal and 128 vertical Divisions

References

[1] – João Pedro Bastos Fula - Underwater mapping using a SONAR

[2] – Virginia Tech Algorithm Visualization Research Group – https://research.cs.vt.edu/AVresearch/hashing/quadratic.php

[3] – Carlos Moreno – https://ece.uwaterloo.ca/~cmoreno/ece250/2012-02-01--hash_tables.pdf

[4] – Wikipedia – https://en.wikipedia.org/wiki/Hash_table#Dynamic_resizing

[5] – Tobias Maier – https://github.com/TooBiased/DySECT

[6] – Joaquín M López Muñoz – https://bannalia.blogspot.com/2022/06/advancing-state-of-art-for.html

[7] – Daniel Lemire, et al. – https://arxiv.org/abs/1902.01961

Slide deck source

Slide deck